Fingerprinting All AI Cluster I/O Without Mutually Trusted Processors
Abstract

1. Introduction

The verification of artificial intelligence use under mutual distrust is an emerging field of technical research and security policy. Much of the research in this area focuses on *regulation, compliance* and *safety* of declared AI systems and deployment. In this work, we tackle a related, yet distinct, problem of *comprehensiveness* – verifying that all traffic leaving a datacenter is explained by the execution of ML workloads declared by a datacenter operator (the 'Prover'). In this scenario, the 'Verifier' could be a regulatory agency, a government, members of an international treaty or any other party that has an interest in preventing illicit uses of AI.

Rather than attempting to prevent covert computation on the Prover's hardware, we instead aim for the communication chokepoint between that hardware and the Prover: the datacenter north-south internet uplink[1]. If the Verifier can capture, hash and timestamp[2] *all* traffic in and out of the datacenter, and the Prover can – upon later request – demonstrate that their declared workloads correspond to this hashed traffic, the verifier can be satisfied that the prover is not obtaining results from undeclared computations in the cluster. Naturally, this raises several important issues:

- How can the verifier be sure that they capture all of the ingress/egress traffic?
- How can we ensure that the cluster does not hide meaningful information in the traffic being monitored? This could be achieved through steganographic techniques, side-channels, timing attacks, etc.
- How can the prover be assured that the border patrol device does not introduce an attack vector on their proprietary secrets?
- Can this solution be retrofitted to realistic, full-scale deployments at the scale of O(100k) GPUs and millions of communication packets streamed in/out every second?

We state our working assumptions here:

- Input steganography via payload data (tokens etc.) is unconstrained.[3] However, the output payload's strict causal dependence on the **declared** inputs is verified post-hoc, based on the evidence captured by the border

[1] Solutions for on-site, out-of-band management and maintenance are out of scope for us.

[2] For security reasons, timestamps can be computed by the verifier and prover upon receiving hashes, rather than inside the border patrol device. Those timestamps are helpful for retrieval from large storage systems later on.

[3] A warden could – in principle – rewrite prompts and input media files while preserving semantics. We leave this open for future research.

patrol device. Only payload data is hashed, as protocol header nonces would otherwise need to be stored long-term[4].

- The Prover needs to store plaintext data long enough for it to be available for verification later.[5]
- Most traffic is API (customer service, language models). Most API traffic is egress.[6]

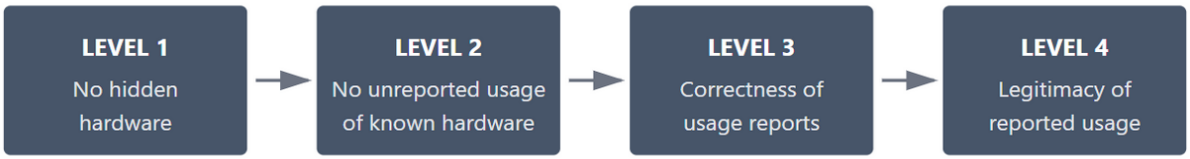
Our main contributions are:

An actionable design specification, feasibility assessment and simulation for our proposed device with strong security properties, which includes the following two features:

1. Hash the payload (layer 7) traffic of all egress and ingress traffic of an AI cluster, and send this evidence data to a verifier, enabling verification of any sample of said traffic later on. These hashes are confirmed to not leak confidential information; more details in *Methods*.
2. Scrub any secret communication channels (by which we mean unverifiable degrees of freedom from the cluster to the prover). This includes analog, timing, and steganography channels[7]. We build on work by Fisk et al. [web, c] and extend it by designing a warden that is observable by two mutually distrusting parties.

Out of scope

In order to convey the nuance of the problems we are –and are not– tackling, we briefly explain the "verification pipeline" we are assuming:



We are specifically aiming for level 2: Capturing raw evidence of computation (only hashes, not plaintext). The hashes, together with the plaintext data it points to, would later feed into a secure cluster trusted by the verifier, without the plaintext information being shown to them directly.

[4]Otherwise, the prover could not satisfy challenges of hashes that were computed with those protocol header nonces (e.g. ISN) included.

[5]This includes protocol layer 7 (application) payload data, if the hashes are only computed at those.

[6]Because of json packaging [web, b] around each streamed token (worst case). Another reason why we focus on egress is working assumption 1.

[7]We specifically refer to non-ML generated steganography channels. The precise verification/replay of ML inference [Cankaya, 2026] is an open research challenge. As a sidenote, we actually considered it among our project ideas for the hackathon.

The verification in said cluster is twofold, and out of scope for us: replaying observed inputs and outputs to confirm they match precisely[8] (level 3), and evaluating the computations for legal compliance (level 4).

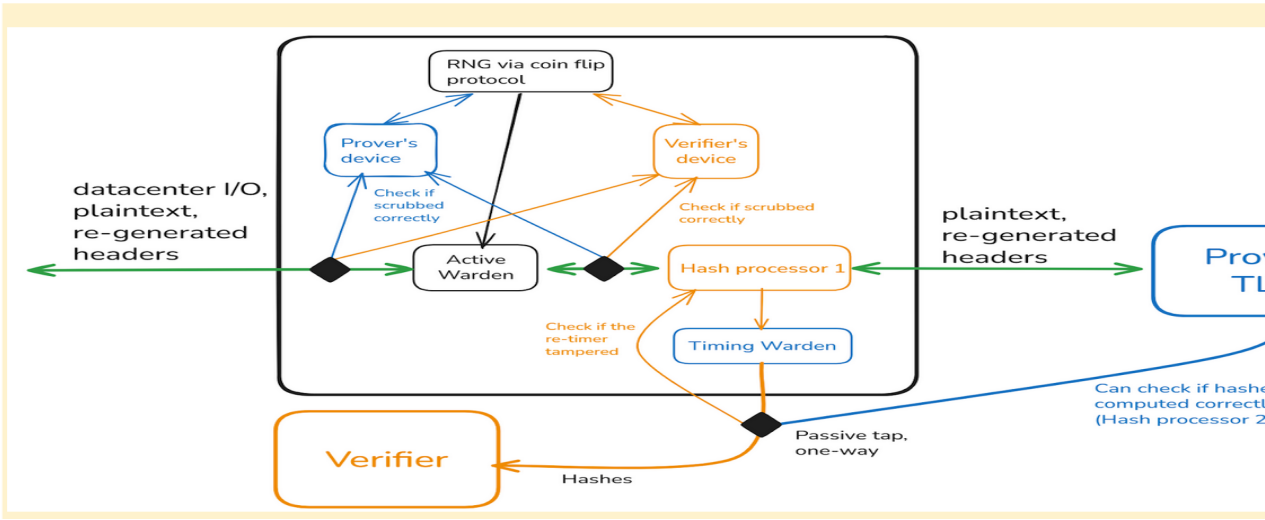
Also out of scope of this work is the physical monitoring of both the prover's cluster and the verification facility needed to prevent secret communication *devices*. Finally, we do not go into detail on steganographic degrees of freedom in unverifiable information contained in application layer payloads (e.g. a limit on GPU temperature diagnostics leaving the facility). These degrees of freedom are not for our border patrol device to erase, but for rules to constrain[9].

2. Related Work

We draw from research by Fisk et al. [**?**] and Xing et al. [pdf, 2026d] on active wardens for erasing steganography channels in protocol headers, as well as Lee et al. [pdf, 2026b] and Uttatwar et al. [web, 2026g] for erasing timing side-channels by adding jitter in a bump-in-the-wire. Both methods are part of our proposed border patrol edge device. While replay verification is not part of our project scope, we refer to Karvonen et al. [**?**] and Cankaya [Cankaya, 2026] for highlighting the importance of this downstream challenge for closing off exploitable degrees of freedom [**?**] in the replay's inaccuracy if not mitigated.

3. Methods

We recommend using our interactive demonstration [web, 2026i] of how our evidence capture device ("border patrol") works. In addition, we give a more detailed treatment of our method and setting below.

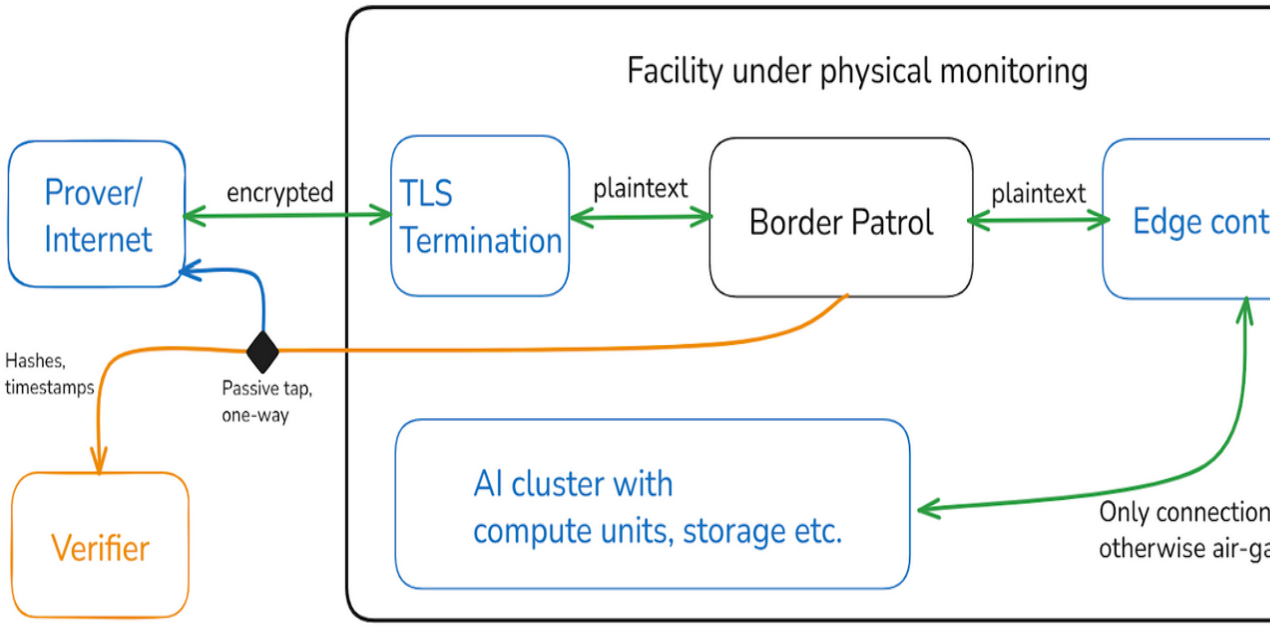


[8] Same as previous footnote.

[9] Since this information is hashed, compliance checking here is trivial.

The most important points about the macroarchitecture (placement of the border patrol device):

1. The facility is under physical monitoring, ensuring that any traffic in and out of the cluster needs to pass through the border patrol chokepoint.
2. Every bit of traffic sent to the verifier (hashes only) can be passively observed by the prover. This places the root of trust in *physics*: Beam splitting via FBT or PLC [web, 2026d] (glass touching glass) and optical isolators [web, 2026e] (one-way-street) ensure the prover can observe and double-check that hashes have been computed exactly as they expect, without being able to interfere with the verifier's monitor leg. Timestamps can be computed by both parties upon receiving the hashes at practically the same time.
3. For unfamiliar readers, TLS is the encryption between the outside world and the datacenter. Unencrypted communication within datacenters is not unusual [**?**]. With key escrows, or retroactive zero-knowledge proofs, or similar, we expect the proposed setup to also be adaptable for working with encrypted traffic, but here we leave this out of scope for simplicity.



The most important points about the microarchitecture (the border patrol device itself):

1. Here we also illustrate the second functionality of the border patrol device: An active warden **without supply chain trust**. While Fisk et al. [**?**] and Xing et al. [pdf, 2026d] have demonstrated the elimination of steganography

channels in protocol headers first in software, then in hardware, both assume single-party trust in the device. In contrast, we combine a coin flip protocol [web, 2026b] and passive observability of the warden's behaviour to *eliminate* the requirement of *mutual* trust in *any* device[10][11].

2. In the illustration above, we colour-code the *minimum* device trust assumptions. This does not mean that transparent design and supply chains could not *also* establish mutual trust in components, which would further add defense in depth.[12]

3. Re-timing eliminates both analog side channels (e.g. light modulation in optical fibre) as well as timing modulation at any relevant bandwidth. Retiming is performed by the Active Warden [**?**] device by buffering all incoming packets (e.g. for 10ms) and then sending them out in their original order at maximum bandwidth to combat attacks such as PHY-Covert Channels [pdf, 2026b].

We present an enhanced design in the appendix section E.
While our proposal is conceptual and tested in simulation rather than real-world conditions, we emphasize that all of the individual components[13] for the border patrol device are **established and field-tested**: Passive optical fiber splitters [web, a], coin flip protocols [Filecoin, 2026], hash functions+timestamps [web, 2026k][14] and supply-chain [pdf, 2026c] hardened [web, 2025] FPGAs [pdf, 2026a] for computing at high line rates [web, 2026a], protocol header scrubbing [pdf, 2026d]/ [web, 2026l] traffic normalization [web, 2026l], and re-timing devices [**?**] against timing and analog side channels.

The particular combination of these components we are proposing is novel, and we simulate their behaviour using software written in Python and available on Github [derpyplops, 2026]. Our simulation works as follows:

1. A server hosts a JSON file (this is intended to replicate a datacenter hosting some output tokens stored in JSON that are intended to be sent to a user that has requested them)

[10]Why is trusted RNG required for the warden? The answer is that some network protocol headers need to include random nonces (e.g. ISNs [web, 2026j] shown at L3) for security reasons, and predictability would break this. The warden overwriting these nonces may exfiltrate information in service of the prover or verifier if not mutually observed or trusted.

[11]To prevent latency penalties, random nonces are not generated on-demand. Instead, the coin-flip protocol populates a pre-computed entropy buffer. The active warden consumes these numbers in a strict First-In-First-Out (FIFO) order to overwrite protocol headers at line rate.

[12]The information path between the verifier and the hashing processor in particular is precarious if the processor is controlled by only them. While the prover *can* observe and double-check this information, photonic or timing modulation on the outputs themselves may be dangerous channels. In appendix E, we present an extended version of the border patrol device designed to defend against such channels.

[13]With the possible exception of the coin flip devices. Since only unilateral trust is required, off-the-shelf CPUs, perhaps TEEs should suffice.

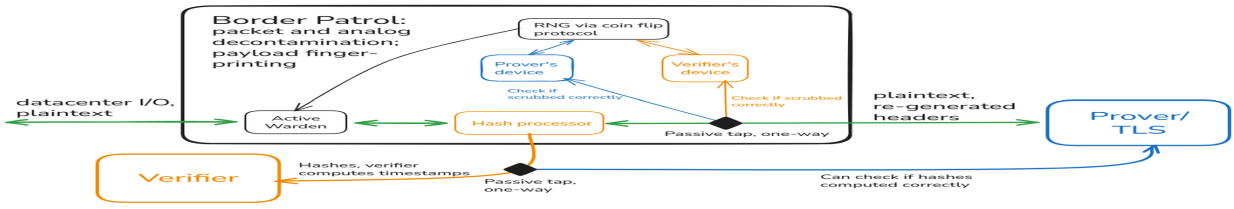
[14]Timestamps can of course be computed after the verifier and prover have received their hashes. We do not expect nanosecond-microsecond precision to be necessary here.

2. The client requests the JSON file (this is the user that is expecting some output tokens from a language model)
3. The server sends the JSON file over either TCP or UDP in a way that makes the packet stream reproducible (this is important because it is what allows the verifier to later confirm that the traffic implies no undeclared work was performed)
    a. Before the packets are forwarded to the client, the warden scrubs all non-reproducible components of the packets and sends them to the client
    b. We additionally construct the TCP and UDP packets to be deterministic, which we do by assembling the packets manually in userspace, where this would typically be done by the kernel using nondeterministic heuristics
4. We log hashes and timestamps for each packet (this simulates the passive tap)

The verifier will later run the exact same procedure on their own hardware, comparing their traffic hashes and timestamps to confirm that the prover's traffic implied they executed the same program as the verifier.

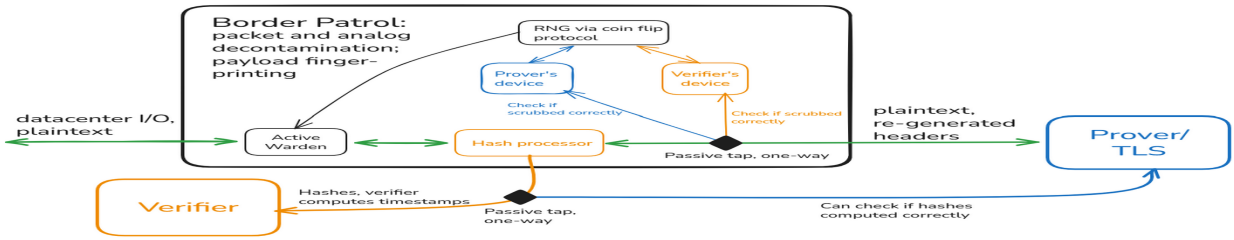
4. Results

For both TCP [web, 2026f] and UDP [control databases GND, 2026] traffic, we are able to detect unexpected bytes in Ethernet frames [web, 2026c]. We simulate this by having the prover send additional Ethernet frames in an attempt to send the client an additional JSON file. After scrubbing the non-reproducible fields from the frames, the comparison is trivial: the verifier compares a hash of its traffic with a hash of the prover's traffic.[15]



The output of analyze.py when we run our simulation using UDP with steganography disabled. The traffic is identical, so the hashes match.

[15]This is somewhat different from the live-checking of the protocol headers via the verifier's and prover's respective devices in the border patrol machine, as shown above. The security of ISNs is not broken here since their generation seeds are kept inside the warden, but not revealed until they expire.

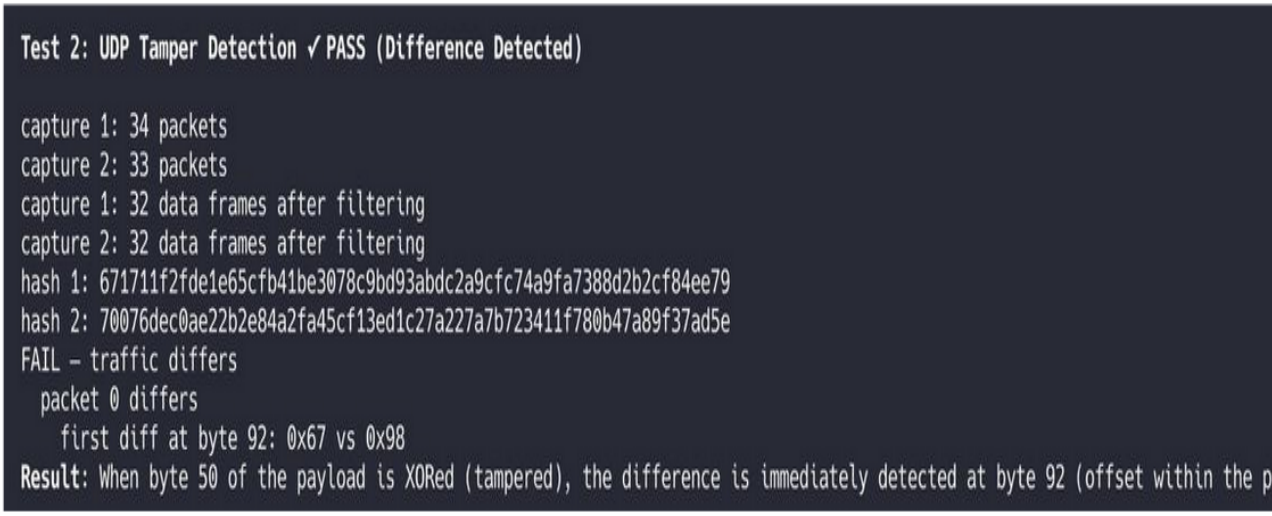


The output of analyze.py when we run our simulation using UDP with steganography enable. The traffic differs, so the hashes are not equivalent.

```
═══════════════════════════════════════════════
Test 7: Covert Timing Channel (Scenario 6, Vulnerable)
═══════════════════════════════════════════════
Expected: 'SECRT' is successfully decoded by the client.
— trial 7 (scenario 6) —
tcpdump: listening on veth-c, link-type EN10MB (Ethernet), snapshot length 262144 bytes
[server] payload: 77592 bytes
[server] waiting for trigger ...
[server] trigger received from 10.0.0.2:5000 with scenario 6
[server] scenario 6: sending 53 UDP frames with covert timing 'SECRT'
[server] covert pattern: 0101001101000101010000110101001001010100
[server] done
./run.sh: line 77: warning: command substitution: ignored null byte in input
[client] scenario 6
[client] using UDP with Timing Analysis
[client] trigger sent
[client] covert analysis: FOUND PATTERN
COVERT_DATA_JSON: {"bits": "0101001101000101010000110101001001010100000000", "secret": "SECRT\u0000"}
[client] decoded secret: 'SECRT'
```

The output of analyze.py when we run our simulation using TCP with steganography enabled. The traffic differs, so the hashes are not equivalent.

Additionally we demonstrated an attack similar to PHY-Covert Channels [pdf, 2026b] to exfiltrate the word “SECRT” using our standard Warden device:



Our steganography simulation is drastically less sophisticated than plausible real-world steganography attempts – the prover has not made any real effort to disguise the unexpected traffic they are sending. However, it is still a demonstration of catching discrepancies in traffic using an active warden and reproducible network traffic. Further, if the traffic is entirely either reproducible or warden-generated, unless the warden is manipulated into rewriting nondeterministic fields with information the prover intends to exfiltrate *and* the verifier device's observation of the warden behaviour fails, steganography is not possible at the protocol level. Therefore, we argue this is a reasonable demonstration of steganography detection.

With our proposed packet-buffering approach enabled we manage to fully divert the attack at timing frequencies of less than 1ms, constraining this attack vector to a max. theoretical bandwidth of 1kbps, while only adding minimal latency (avg. inter packet delay of 0.01ms/packet) and throughput costs (-3.4%).

```
--- Test 8: Covert Channel Timing Analysis (Secure - Fuzzy Barrier) ---

Analyzing timing for: /data/capture_8.pcap
Packet Count: 53
Mean IPD: 0.3012 ms
Variance: 0.210052 (scaled)
Estimated Bandwidth: 3319.89 bps (raw)
Distribution:
  ~200us: 0
  ~500us: 0
  ~1ms:   9
RESULT: Indeterminate / Noise


Analyzing timing for: /data/capture_7.pcap
Packet Count: 53
Mean IPD: 0.3113 ms
Variance: 0.019479 (scaled)
Estimated Bandwidth: 3212.04 bps (raw)
Distribution:
  ~200us: 35
  ~500us: 16
  ~1ms:   0
RESULT: Modulated (Timing Channel Detected) - VULNERABLE
```

5. Discussion and Limitations

Limitations

- We only achieve a simulated bandwidth of approx. 39 Mbps, so it's not clear whether a software approach for implementing the border patrol device is feasible, possibly necessitating significant hardware design challenges Still, we presented strong precedents for the individual components, and active wardens have been built to handle 100G line rates [pdf, 2026d], which would in fact be over-provisioned for most long-range internet fiber connections.
- We do not constrain bandwidth for covert communication *into* the datacenter. While the input data has many uncontrollable variables (media input, user prompts, . . .), input wardens may re-write such information while preserving semantics. We leave this to future work.
- There still is some potential for covert outbound communication via low-frequency timing modulation, but many orders of magnitude less than without the border patrol device.

Future Work

- A hardware-implementation demo of our proposed setup. We expect the bill of materials of a somewhat representative device to cost less than the researcher's and engineer's time needed to build it.
- Threat modeling for very low outbound bandwidth–but still impactful–covert workloads in the datacenter.

6. Conclusion

Firstly, a brief feasibility assessment: In the appendix B, we calculate a bandwidth of ~40GB/s API egress out of a 100.000 GPU AI cluster streaming 200.000.000 tokens per second. The border patrol device, even under such extreme assumptions, can use off-the-shelf hardware. The storage requirements for both the prover and the verifier are insignificant for any realistic data management solution. Secondly, we see passive optical splitters as under-appreciated assets for designing verification setups under mutual distrust. Finally, we encourage further work on the hardware development to prepare border patrol devices for AI governance quickly, as they may be needed soon.

Code and Data

- **Code repository**: https://github.com/derpyplops/warden [derpyplops, 2026]
- **Interactive demo link 1:** https://warden-hackathon.vercel.app/ [web, 2026m]
- **Interactive demo link 2:** https://www.felix-krueckel.com/warden.html [web, 2026i]
- **Ethernet protocol visualization:** https://ethernet-frame-visualization.vercel.app/ [web, 2026h]

Author Contributions

Naci Cankaya led the project and architecture design of the border patrol machine. Jakub Krys led the red-teaming of the security and feasibility via thought experiments and Fermi estimates. Jonathan Ng led the creation of the simulation codebase. Felix Krückel contributed the visualizations and literature research into covert communication threats. Luke Marks contributed insights into Ethernet protocol details. All authors contributed in general literature research, fact-checking and writing.

Appendix A: Prover's Verification Mechanism

We now consider the problem of how verification of such hashed values could take place. The fundamental challenge is as follows: given a hash value chosen by the Verifier out of the annual pool of hashes, how can the Prover reliably and efficiently demonstrate that they possess a record entry which hashes to this value[16]?

A simple idea is to maintain a traditional 'hash table' – a dictionary of all plaintext LLM inputs, the corresponding plaintext outputs and their hash values.

[16]Once again, in this work we do not consider the problem of verifying that this declared workload is compliant with some regulations, only the problem of verifying that no other undeclared workloads have been run.

Since the Prover already controls the egress from the data centre, they can easily register how many Ethernet frames each output gets broken into. Furthermore, since the Border Patrol protocol would be fully open-source, they also know the 'hashing boundary' – how many frames are included in one hash. This allows the Prover to have a causal association between a prompt, the plaintext output and the corresponding sequence of hashes. The hash values flow back to the Prover through a passive network tap placed on the connection which carries them from the Border Patrol to the verification cluster (see Fig. 2).

This in turn can be used for an extremely efficient hash lookup. When the Verifier specifies the hash whose provenance is to be demonstrated, the Prover finds the key associated with this hash and forwards it to the secure verification facility for replay. The facility recomputes the hash and presents it to the Verifier, who is then satisfied that the Prover must have carried out a workload that produces this hash.

The Border Patrol protocol needs to agree on the number of Ethernet frames corresponding to the fundamental 'verification unit' that gets hashed. This number is arbitrary and should strike a balance between storage requirements and recomputation time. At the extreme end of the spectrum, we could hash every single Ethernet frame, leading to the highest possible number of hashes in storage. However, since in this scenario one hash corresponds to a very small number of output tokens, this minimises the number of outputs that need to be recomputed in the verification facility. On the other hand, we could hash together all Ethernet frames that pass the Border Patrol within a month, which would lead to only one hash in storage, but would require one month of recomputation during verification.

The Prover needs to maintain knowledge of which inputs and parts of the corresponding outputs were included in each hash. For example, assume that the hashing window is 150 Ethernet frames, two input prompts generate 120 and 70 tokens, respectively, and one output token is streamed out via a single Ethernet frame. The Prover must then associate the following with the hash values h1 and h2:

```
{
h1: [(prompt_1, output_1, 1, 120), (prompt_2, output_2, 1, 30)],
h2: [(prompt_2, output_2, 31, 70), ...],
...
}
```

In this way, when the Verifier demands that hash h1 be reproduced, the Prover can immediately look up the fact that prompts 1 and 2 must be sent out to the verification facility[17] (alongside the corresponding outputs and their indices).

[17]Note that in general, these could be batched prompts or some other type of input. We

This information is necessary for the verification facility to faithfully reproduce the exact stream of 150 Ethernet frames that should be hashed and compared to h1.

Appendix B: Storage Requirements

We start by calculating an upper estimate of the volume of the hashes and plaintext data that would be required to be stored by the verifier and prover respectively. We assume worst-case conditions in order to demonstrate the feasibility of our proposal.

First, let us assume that the datacenter under monitoring contains 100k GPUs of the NVIDIA Hopper generation. Each such GPU is assumed to be capable of a 2000 token/s inference throughput[18], meaning that the total throughput of the cluster is 2*10^8 tokens/s. Next, we need to calculate how many bytes of data this token volume corresponds to. For this, we assume that the tokens are wrapped in OpenAI's streaming API template of the following format:

“‘

{"id":"chatcmpl-123","object":"chat.completion.chunk","created":1694268190,"model":"gpt-4o-mini", "system_fingerprint": "fp_44709d6fcb", "choices":[{"index":0,"delta":{"content":"Hello"},"logprobs":null,

“‘

Importantly, note that this template includes a single response token only. This is because the streaming API aims to deliver generated tokens to the user as quickly as possible, rather than waiting for the full generation to finish before sending the payload. The 'one-token-per-json' scenario represents the most challenging configuration for our methodology, as this will lead to the highest possible number of hash computations.

Furthermore, we operate under an (unrealistic) assumption that each such template is allocated to one Ethernet frame. The payload is around 200 ASCII characters (see the streaming format above), corresponding to 200 bytes (+ headers) – well below the Maximum Transmission Unit (MTU) of the Ethernet protocol.

Therefore, the total number of Ethernet frames streamed out of the cluster is simply 2*10^8 per second, same as tokens. We then calculate the SHA-256 hash of each such frame, leading to 2*10^8 * 24 * 3600 * 365 = 6.3*10^15 hash computations performed in a year, with a corresponding hash volume of 6.3*10^15 * 256 / 8 = 2*10^17 bytes ~ 200PB.

Additionally, the Prover needs to maintain a year-long record of all plaintext data transmitted from their cluster to the outside world. Each token generated in their cluster is assumed to be wrapped in the template above. Since one

used single prompts for simplicity in this illustrative example.

[18] Deepseek's inference setup for their v3/R1 models is reported to achieve a token throughput of ~14.8k tokens per H800 node (8 GPUs, decode unit) per second.

ASCII character is stored by exactly 1 byte, this means that to stream one token out of the cluster, 200 bytes are required. On an annual scale, this corresponds to 2*10^8 * 200 * 24 * 3600 * 365 = 1.26*10^18 bytes ~ 1,260PB.

The cost of storing such a large amount of data is small compared to the costs of AI datacenters at the scale considered here (<1%): A typical price for a 28TB HDD hard drive is around 500€[19], thus the full cost amounts to 1,260 PB/28TB * 500€ ~20,000,000€.

Note that all these estimates represent a pessimistic upper bound that arises from *extremely* inefficient data management. When using non-streaming APIs, multiple tokens are included as part of one chat completion template. Further, multiple such templates could be transmitted as part of a single Ethernet frame, as long as the total payload size does not exceed the standard Ethernet MTU of 1500 bytes. Storing API json files for each token is also likely unnecessary when they can be reconstructed from tokens and heuristics. We did not include data compression at all.

Finally, we do not need to compute the hash of each Ethernet frame. Instead, the 'fundamental unit' of verification could be an arbitrarily long chain of frames. All these modifications would reduce storage by >3 orders of magnitude.

Appendix C: Preventing dictionary attacks

The Prover might be worried that the Verifier can deduce the plaintext content of the messages passing through the Border Patrol by guessing which content hashes to the observed hash values. Calculating this content by inverting the hash is not possible – hash functions are by design irreversible. However, an adversarial Verifier could be tempted to use so-called dictionary attacks. In other words, they could pre-compute the hashes of the most common plausible inputs and simply compare them to the observed ones. The feasibility of this approach is entirely determined on the expected size of the input space of the hash function. For example, if there can only ever be 10 possible inputs, then it is trivial to pre-compute their 10 corresponding hashes and infer the input that has produced the hash coming out of the Border Patrol.

Unfortunately for the Prover, the input space of their API responses is indeed somewhat limited: these can only be tokens generated by the LLM[20]. These in

[19] As of early 2026

[20] The surrounding chat completion template that encompasses the output tokens is also a part of the content that gets hashed. However, there are no degrees of freedom there. To understand why, first note that the verification cluster replays the workload using the provided inputs, inserts the output tokens into the completion template, hashes this JSON object and compares it to the observed hash from the Border Patrol. However, the replayed computation will have a different timestamp and fingerprint to the one that produced the original hash. Thus, the Prover needs to supply the verification cluster not only with the input prompt, but also the exact metadata in the output template. In other words, the Prover declares: 'Here is the right prompt and the corresponding template metadata, but not the output itself'. Thus, the only degree of freedom is really limited to the output tokens.

turn are constrained by the size of the LLM vocabulary, typically around 200 000 tokens. Thus, in the scenario presented above – where only one output token is inserted into the streaming completion template, this template corresponds to one Ethernet frame and later one hash – the attacker could pre-compute the hashes of all tokens inserted into these templates. They would then simply perform the lookup in a hash table with 200 000 values and immediately infer the right token. Thus, it would be trivial for the Verifier to guess all the tokens from the streaming API responses.

To calculate how many tokens need to be inserted into the chat template to prevent such attacks through combinatorial complexity, let us consider how many hash computations a motivated attacker could perform in a month. Specialised chips can achieve throughputs of 200TH/s (terahashes per second). Within a month, this corresponds to 5.2*10^20 hashes computed. On the other hand, a sequence of n output tokens out of a 200k vocabulary can lead to (2*10^5)^n distinct inputs. Thus, the attacker is guaranteed to guess the sequence corresponding to a given hash if the sequence is shorter than $log_{2*10^5}(5.2 * 10^{20}) \sim 3.9$ tokens long. Running multiple specialised chips in parallel will not help the attacker much – with 1000 hash-crunching chips, the token number increases to 4.5.

Overall, to be on the safe side, the streaming API template should enclose at least 5 tokens in order to prevent brute-force hash guessing by the Verifier. However, this back-of-the-envelope calculation assumes that the attacker needs to try all possible n-gram combinations of tokens from the vocabulary. This is a totally unrealistic assumption, as the tokens streamed out of the cluster will almost always be semantically meaningful – sequences of tokens that form sentences, code snippets, equations, or similar. Thus, the search space is massively reduced by noting that only a small fraction of the vocabulary is likely to follow a token like 'The'. Assuming that only 1000 most likely tokens need to be checked, our 'effective vocabulary' is reduced to 1000. Then, the calculation becomes $log_{2*10^5}(5.2 * 10^{20}) \sim 7$. In other words, any semantically meaningful sequence of fewer than 7 tokens could be brute-forced by the Verifier in less than a month with a single hash-crunching chip. Thus, to prevent this, the streaming API output template could wrap at least 10 tokens. This should have a negligible effect on user experience – instead of seeing tokens appear on their screen one at a time, the user will receive them 10 at a time. Alternatively (or additionally), the hashes could simply be computed on more than one JSON template or more than one Ethernet frame. For example, if the hashing boundary is set every 100 Ethernet frames, the whole problem disappears even with 1 streaming token per JSON template.

Appendix D: Eliminating steganography needs connection tables

The need to prevent steganographic communication between the data centre and the outside world introduces the need to eliminate certain degrees of freedom

from the protocol metadata. To illustrate this more concretely with an example, in this Appendix we justify why source and destination ports need to be explicitly handled by the Active Warden.

Network protocols grant both communication parties control over certain protocol fields. In UDP, the initiating party (client) selects its source port, whilst the responding party (server) must use the destination port specified in the request as its source port for the response. Without intervention, either party could encode information through port selection.

Consider a simple example: an external client initiating API requests to a datacenter. By choosing source port 5000 versus 5001, the client transmits one bit of information per connection. At 1000 requests per second, this yields 125 Bps or ~10 GB/day of covert channel capacity. Similarly, when the datacenter initiates outbound connections (e.g., fetching training data), it could encode information in its choice of ephemeral source port (typically ranging from 32768-60999, providing ~15 bits per connection). The Active Warden must eliminate both channels.

Two separate connection tracking tables are required, because connections can be initiated from either direction:

- **External-initiated connections** (e.g. API requests to the datacenter) allow an external client to encode information in their choice of client source port. The warden must replace this with a random value, then maintain state to translate responses back to the original port.
- **Datacenter-initiated connections** (e.g. fetching external datasets) allow the datacenter operator to encode information in their choice of client source port. Again, the warden must replace this with a joint random value and maintain a reverse translation state.

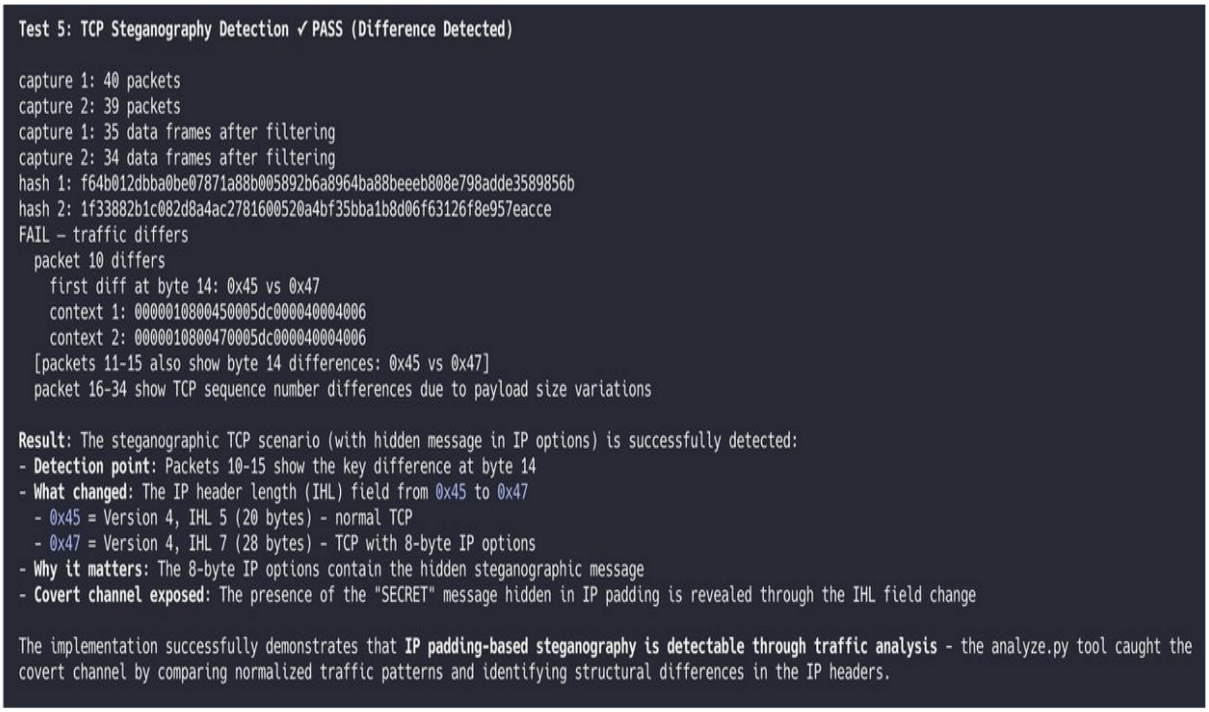
These cannot share a single table because the same port number (for example, 5001) might be simultaneously assigned to a datacenter client port in one connection and to an external client port in another. The tables must be keyed by the full connection 4-tuple: (source IP, source port, destination IP, destination port) in order to disambiguate these two potentially conflicting cases.

Each state-tracking table has two distinct security functions during the connection lifecycle. First, during the request phase, the table performs port randomisation: the initiating party's source port (client port) is replaced with a value from the RNG, eliminating their ability to encode information. Simultaneously, the table records the target server port from the request's destination field. During the response phase, the table performs enforcement and back-translation: it verifies the responding party uses the correct server source port (which prevents the server from encoding information through port choice), and it translates the response destination port from the RNG value back to the original client port. This ensures that the response is delivered to the correct client port and does not carry covert information through the choice of the server port.

Similar translation tables need to be maintained wherever parts of the relevant network protocol offer the attacker degrees of freedom that could be used for steganographic communication. This could introduce significant programmatic complexity for more complicated protocols like TCP, which are stateful. In TCP, the Active Warden must additionally perform sequence number translation on every single packet, and in both directions. More precisely, during the so-called ‘TCP handshake’ the client's Initial Sequence Number (ISN) is replaced with a random number. Thus, the Warden must record the offset between the original and scrubbed ISN, so that all subsequent packets in this connection have their outbound sequence numbers incremented by this offset. Similarly, the corresponding inbound ACK numbers in this connection need to be decremented by the same offset. This is necessary to maintain consistency in both the original and scrubbed sequences. Overall, the Active Warden must track connection state to handle the handshake phase, data transfer phase, and connection termination. Each TCP connection might require around 200 bytes of state-tracking data. For a data centre with O(1000) concurrent connections processing millions of packets per second, the Active Warden must implement efficient hash lookups to avoid slowdowns in serving this traffic to customers.

Appendix E: Enhanced design for the border patrol device

The architecture shown below combats timing and analog modulation side-channels between the verifier and their hash processor in the border patrol device. We refer to Uttarwar et al. [?] for more details on the timing warden. Another enhancement here is passive observation on both sides of the active warden.



LLM Usage Statement

We used Claude, Gemini and ChatGPT to explain concepts and look for source materials. Claude Code was used to help with creating the simulation of the proposed methodology, and Gemini for the accompanying visualisation websites.

Limitations & Dual-Use Considerations

- Limitations (false positives/negatives, edge cases, scalability constraints)
  - False positives may be triggered by bit-errors in hashing processors, leading to unverifiable hashes even with re-execution. Redundant hash computation in the border patrol device can help combat this if needed.
  - Potential edge cases include extreme file upload and download bandwidths in and out of the cluster, beyond anything needed for orchestrating workloads or customer service.
  - Multi-datacenter training requires extreme-bandwidth connection in between. Any such connected group would need to be physically isolated.
- Dual-use risks (could your method be used to train better manipulators?)

  - We do not think so. The protocol relies on transparency and assumes both parties have full knowledge of the border patrol device's inner workings.

- Responsible disclosure recommendations (if vulnerabilities discovered)
  - We have not discovered novel vulnerabilities in any system.
- Ethical considerations in your approach
  - We do not see any concerns here.
- Suggestions for future improvements
  - See "future work" above.

References

Passive optical fiber splitters, a. URL https://www.keysight.com/au/en/products/network-visibility/network-taps/flex-tap-fiber-optical.html. Accessed from keysight.com.

json packaging, b. URL https://platform.openai.com/docs/api-reference/chat/create. Accessed from platform.openai.com.

Fisk et al., c. URL https://www.researchgate.net/publication/2550303_Eliminating_Steganography_in_Internet_Traffic_with_Active_Wardens. Accessed from researchgate.net.

Protection against supply chain attacks - sdot secure network card, 2025. URL https://www.infodas.com/en/solutions/sdot-secure-network-card/. Accessed from infodas.com.

Infodas URL https://www.criticalcommunications.airbus.com/sites/g/files/jlcbta221/files/2025-06/Infodas%20SDoT%20SecurityGateway%20Flyer.pdf. PDF document.

nsdi14 paper lee, 2026b. URL https://www.usenix.org/system/files/conference/nsdi14/nsdi14-paper-lee.pdf. PDF document.

overview supply chain wp, 2026c. URL https://ww1.microchip.com/downloads/aemdocuments/documents/fpga/ProductDocuments/SupportingCollateral/overview_supply_chain_wp.pdf. PDF document.

sec20 xing, 2026d. URL https://www.usenix.org/system/files/sec20-xing.pdf. PDF document.

Ia-780i fpga accelerator with intel agilex 7 fpga, 2026a. URL https://www.bittware.com/products/ia-780i/. Accessed from bittware.com.

Cryptographic coin flips: Commitment schemes, mpc, and verifiability, 2026b. URL https://coinfliptool.com/articles/cryptographic-coin-flips. Accessed from coinfliptool.com.

Ethernet frame - wikipedia, 2026c. URL https://en.wikipedia.org/wiki/Ethernet_frame. Accessed from en.wikipedia.org.

Fiber-optic splitter - wikipedia, 2026d. URL https://en.wikipedia.org/wiki/Fiber-optic_splitter. Accessed from en.wikipedia.org.

Optical isolator - wikipedia, 2026e. URL https://en.wikipedia.org/wiki/Optical_isolator. Accessed from en.wikipedia.org.

Transmission control protocol - wikipedia, 2026f. URL https://en.wikipedia.org/wiki/Transmission_Control_Protocol. Accessed from en.wikipedia.org.

Uttatwar et al., 2026g. URL https://www.epj-conferences.org/articles/epjconf/pdf/2025/13/epjconf_icetsf2025_01049.pdf. Accessed from epj-conferences.org.

Warden: Protocol stack explorer, 2026h. URL https://ethernet-frame-visualization.vercel.app/. Accessed from ethernet-frame-visualization.vercel.app.

Fingerprinting ai cluster usage, 2026i. URL https://www.felix-krueckel.com/warden.html. Accessed from felix-krueckel.com.

Warden: Protocol stack explorer, 2026j. URL https://www.felix-krueckel.com/tcp-ip-visualizer.html. Accessed from felix-krueckel.com.

Rfc3161 compliant time stamp authority (tsa) server, 2026k. URL https://knowledge.digicert.com/general-information/rfc3161-compliant-time-stamp-authority-server. Accessed from knowledge.digicert.com.

Openbsd pf: Packet filtering, 2026l. URL https://www.openbsd.org/faq/pf/filter.html. Accessed from openbsd.org.

Verifying agreements on compute use using trusted networking hardware, 2026m. URL https://warden-hackathon.vercel.app/. Accessed from warden-hackathon.vercel.app.

Naci Cankaya. Catching misreporting about ml hardware use by turning noise into signal, 2026. URL https://nacicankaya.substack.com/p/catching-misreporting-about-ml-hardware. Accessed from nacicankaya.substack.com.

Authority control databases GND. User datagram protocol - wikipedia, 2026. URL https://en.wikipedia.org/wiki/User_Datagram_Protocol. Accessed from en.wikipedia.org.

derpyplops. derpyplops/warden, 2026. URL https://github.com/derpyplops/warden. GitHub repository.

Filecoin. Filecoin features: Distributed randomness and leader elections, 2026. URL https://filecoin.io/blog/posts/filecoin-features-distributed-randomness-leader-elections. Accessed from filecoin.io.

Gina Fisk. Eliminating steganography in internet traffic with active wardens, 2026. URL https://digitalcommons.memphis.edu/facpubs/2766/. Accessed from digitalcommons.memphis.edu.

Adam Karvonen, Daniel Reuter, Roy Rinberg, Luke Marks, Adrià Garriga-Alonso, and Keri Warr. Difr: Inference verification despite nondeterminism, 2025. URL https://arxiv.org/abs/2511.20621.

mbender ms. Enabling end to end tls on azure application gateway, 2026. URL https://learn.microsoft.com/en-us/azure/application-gateway/ssl-overview. Accessed from learn.microsoft.com.

Roy Rinberg, Adam Karvonen, Alexander Hoover, Daniel Reuter, and Keri Warr. Verifying llm inference to detect model weight exfiltration, 2025. URL https://arxiv.org/abs/2511.02620.

Vrushali Uday Uttarwar and Dhananjay M. Dakhane. Delay normalization technique to disrupt covert timing channels using active warden | epj web of conferences, 2026. URL https://www.epj-conferences.org/articles/epjconf/abs/2025/13/epjconf_icetsf2025_01049/epjconf_icetsf2025_01049.html. Accessed from epj-conferences.org.